CS267 HW2-1 Write Up

Contributions:

- Write-Up: All
- Brandon: Serial optimization design, implementations, and graphing; Computation v synchronization
- Kevin: OpenMP implementation, Further Serial Optimizations, Graphing and benchmarking parallel

Introduction:

Homework 2.1 focuses on optimizing a 2D particle simulation governed by a basic force equation for repulsive forces. The first objective was to optimize the provided naïve serial solution, which originally implemented an inefficient O(n^2) complexity by computing forces between every pair of particles. The goal was to reduce this to an O(n) complexity implementation by leveraging spatial binning. The second objective was to parallelize this optimized solution using OpenMP, following a shared-memory parallelization approach.

Serial Optimization Approach:

The initial code implementation used a brute-force method that iterated over every pairwise particle interaction, calculating forces for all particles regardless of relative distance. This approach required a two-level nested loop, making it inefficient as it computed forces even for atoms far beyond the interaction range. As a result, the brute-force computation scaled quadratically O(n^2), significantly increasing computational cost as the number of particles grew. To improve performance, we implemented a gridded simulation space, dividing the computational space into bins. This binning strategy allows us to reduce the number of operations by only considering particles within a reasonable range of one another, rather than performing unnecessary calculations for every possible pair.

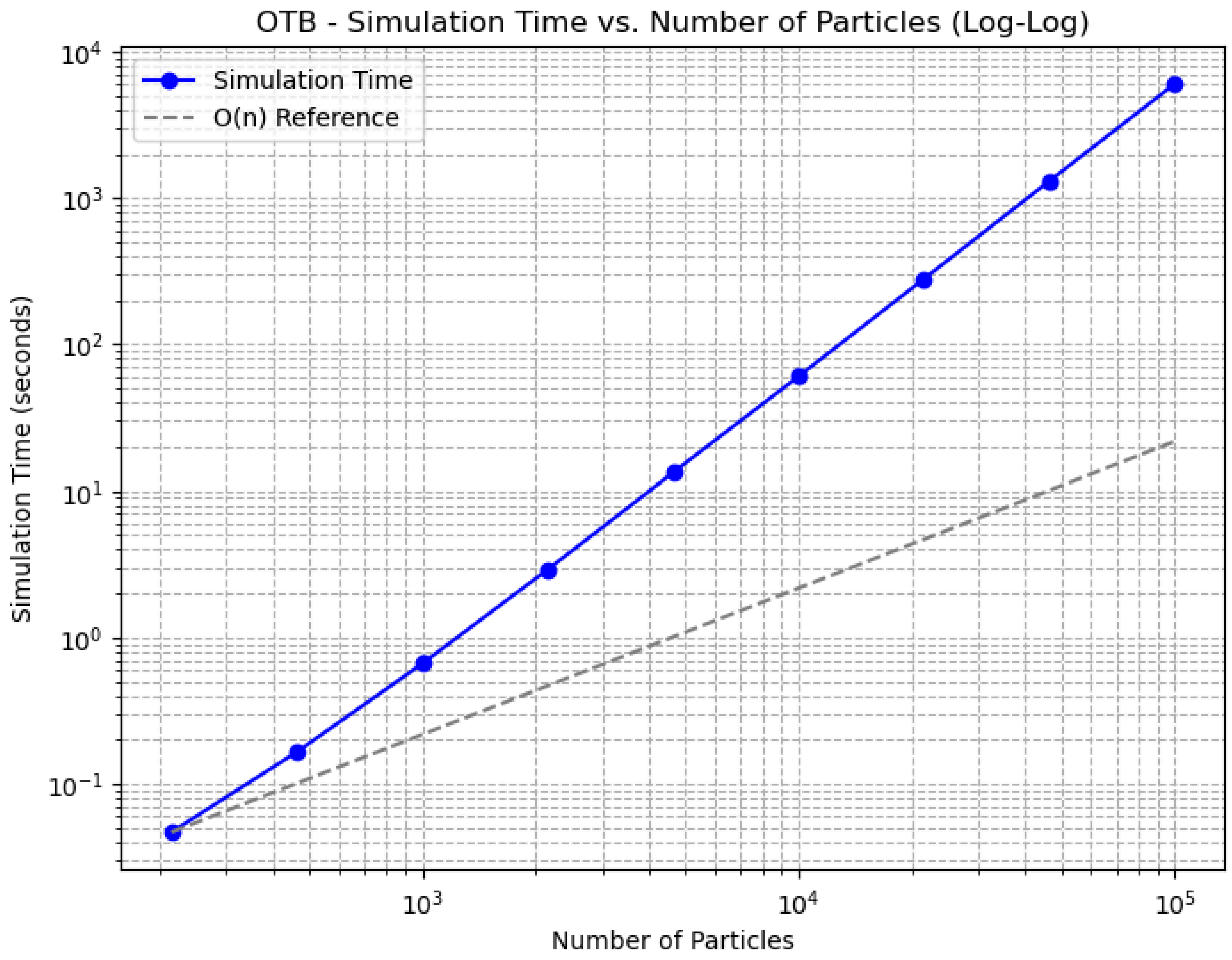
By dividing the space into bins, each containing a subset of particles, we ensure that only particles within the same bin and adjacent bins are considered for force computations. If implemented correctly, each bin should contain an approximately equal number of particles, assuming a uniform density across the simulation. Consequently, instead of growing exponentially with the number of particles, the computation now remains linear as the number of operations per particle is effectively constant.

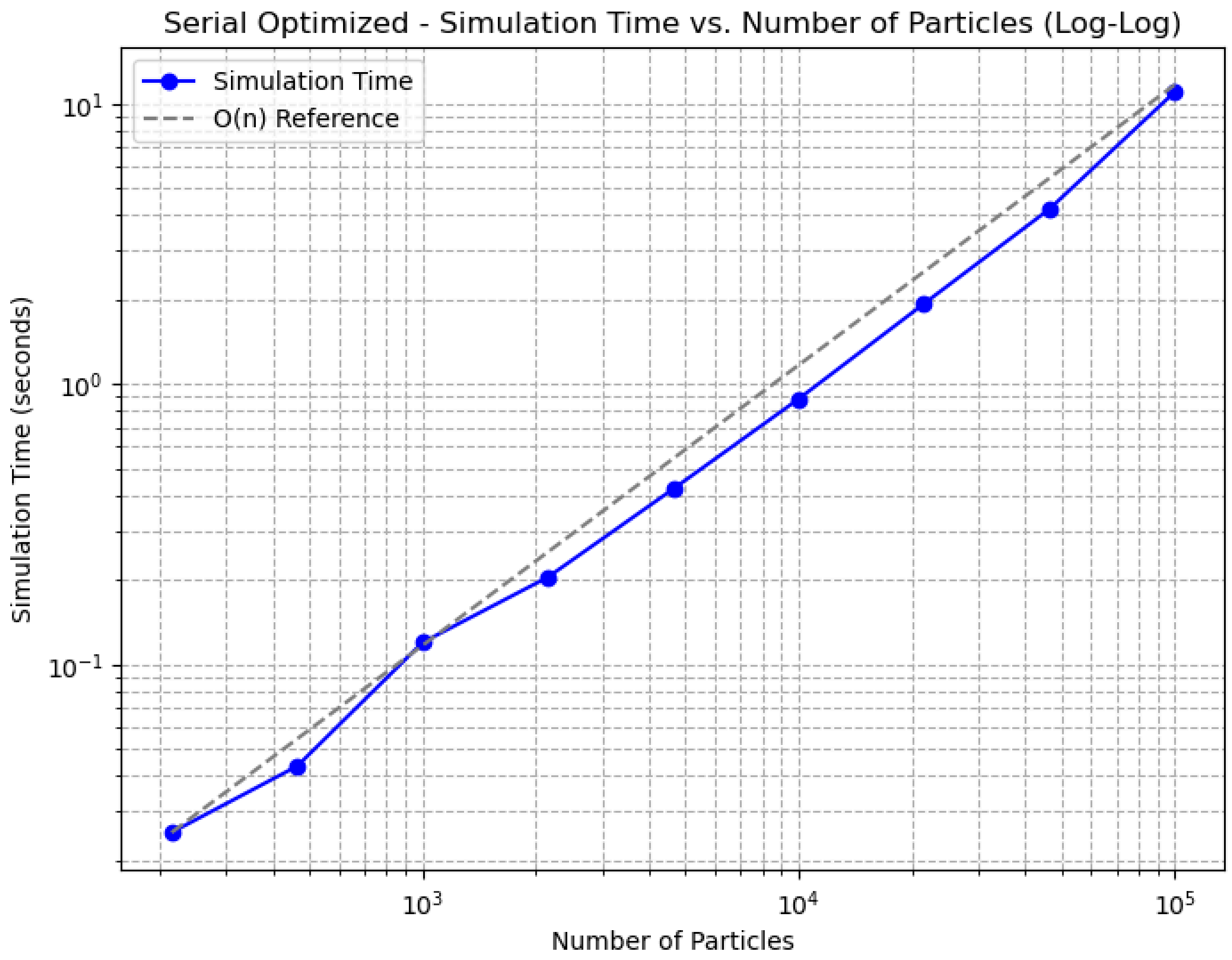
To implement this optimized approach, two key data structures were introduced, along with an initialization stage to set them up and a systematic method for assigning particles. The Bin() struct was created to represent a vector of particle_t pointers, grouping particles by their proximity at any given time. The simulation grid was then constructed using 2D arrays of bins, aligned with the simulation's x and y dimensions. This design choice allowed for efficient spatial indexing and reduced redundant computations.

The initialization process involves assigning particles to bins at t=0, ensuring that the simulation begins with an efficient data structure in place. At each subsequent time step, particles are reassigned only if they move to a different bin. The function get_bin_index() is responsible for determining a particle's bin location, facilitating smooth and efficient bin-to-bin movement tracking.

A key enhancement in our approach was the use of double buffering with current_frame and next_frame vector pointers. These two frames store particle data separately, allowing for clean memory management between iterations. During each time step, forces are computed using the current frame, while updated positions are stored in the next frame. This separation ensures that force calculations remain consistent while simultaneously preventing memory conflicts, an essential feature when implementing parallelization in OpenMP. The ability to swap frames efficiently at each iteration not only optimizes serial performance but also enhances thread-safe access during parallel execution.

The performance improvements from this optimization are clearly reflected in the simulation time graphs. The first graph represents the naïve implementation, demonstrating quadratic scaling as expected, where simulation time increases rapidly with the number of particles due to redundant computations. In contrast, the second graph shows the optimized serial implementation, which closely follows the O(n) reference line. The linear scaling validates that our binning approach effectively reduces computational complexity, making large-scale simulations far more feasible.





With the serial implementation optimized, this restructuring lays the foundation for the second objective: parallelizing the simulation using OpenMP. The spatial binning approach naturally lends itself to parallel execution since each bin can be processed independently. In the next phase, we expect the simulation time to scale inversely with the number of processors (T/p), as force computations are distributed across multiple threads. Proper handling of shared memory access and efficient thread synchronization will be critical in achieving near-optimal speedup.

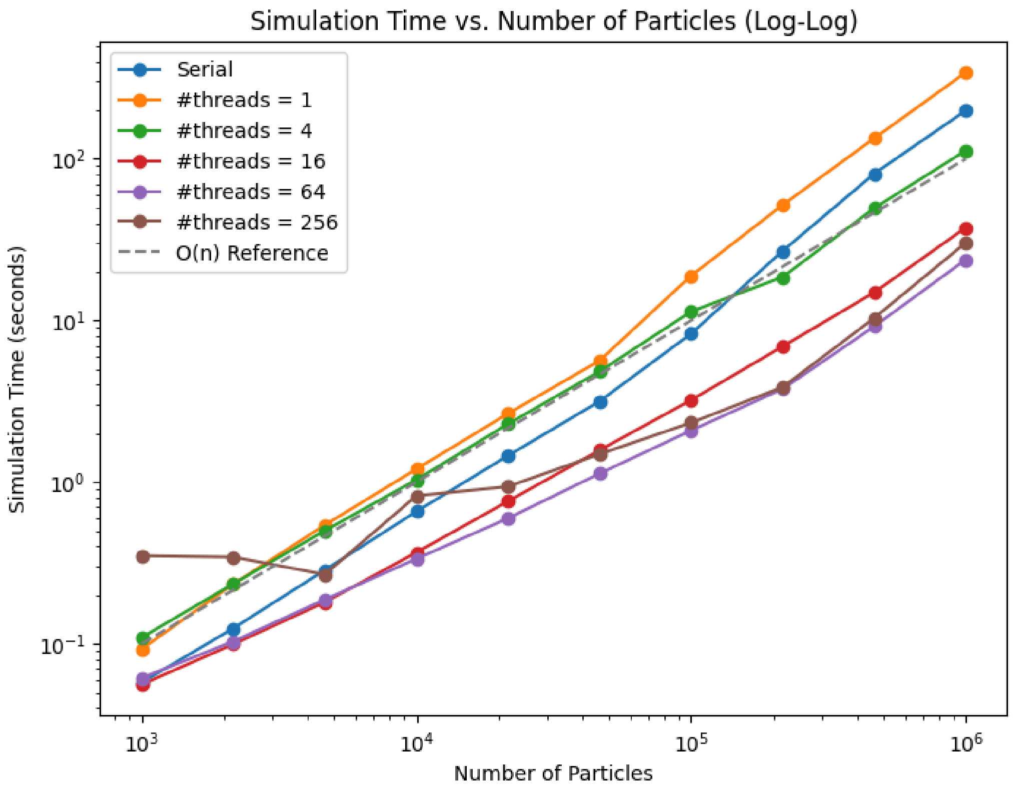
By implementing spatial binning and double buffering, we successfully reduced the computational complexity from O(n^2) to O(n), achieving a significant performance boost for large-scale simulations. This optimization not only improves runtime efficiency but also ensures a scalable framework for parallel execution in future implementations.

Parallel Implementation:

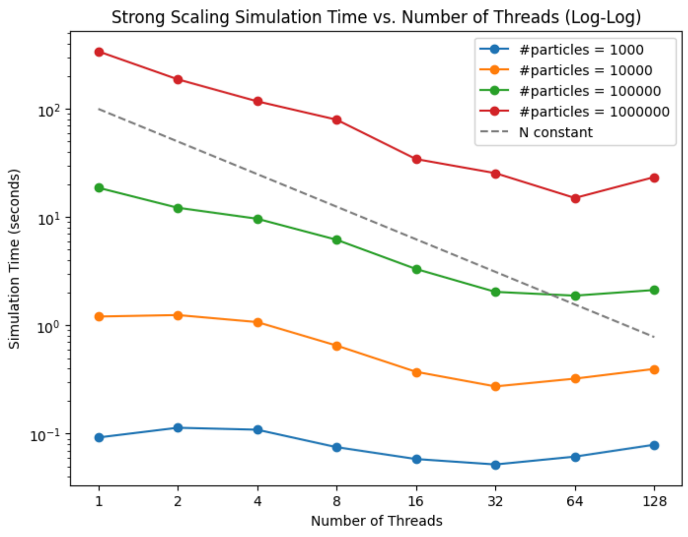
To approach the desired *T/p* speedup, we added pragma directives to parallelize computations across bins. Specifically, we split the work of computing forces, clearing the next frame, moving particles, and reassigning particles to bins across threads. These are all segments in the code where the serial implementation already iterates through bins, so we simply add the #pragma omp for directive to these loops. Computing forces, clearing the next frame, and moving the particles require no synchronization between threads because they do not involve any operations that write to bins. Furthermore, two threads will never write to the same particle, so we do not have to consider data races when updating particles.

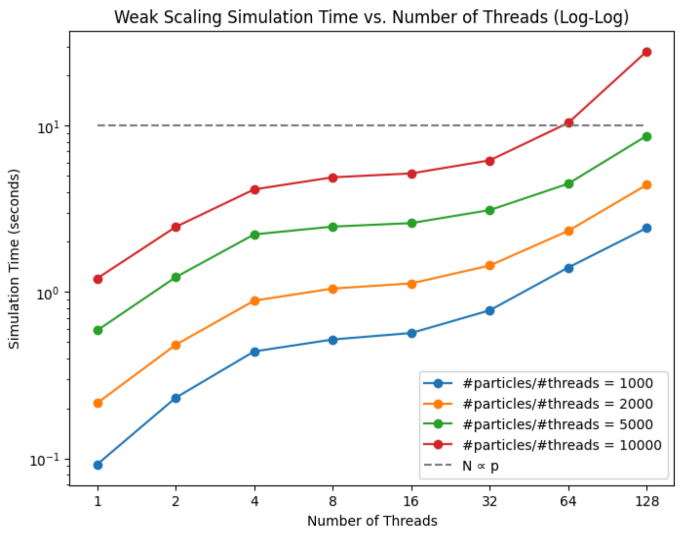
However, reassigning particles to bins may cause data races when particles move across bin boundaries. In this case, the thread will have to write to a bin that it is not operating on, which another thread may access simultaneously if a particle in its bin moves across boundaries into the same bin. To address this, we add a lock to each bin that a thread must acquire before writing to that bin. Thus, the synchronization time of our parallel implementation boils down to how much time these threads spend waiting to acquire locks.

Finally, we add the #pragma omp single directive to the operation that swaps the current and next frames to prepare for the next iteration. This operation only needs to be performed once per iteration, so the directive ensures that only one thread runs it.



The above plot displays a comparison between simulation times for the serial implementation, parallel implementation with a varying number of threads, and the O(n) reference line. For 16 and 64 threads, there is a general improvement in simulation time over the serial implementation across the range of particles tested. Notably, the parallel implementation running on one thread is strictly worse than the serial implementation. This direct comparison showcases the overhead introduced by OpenMP when managing the parallel execution environment. Additionally, there is virtually no improvement in runtime for the parallel implementation when the number of particles is small. In this region of N–near 1000 particles–the parallel speedup is negated by the overhead introduced by thread creation, execution, and synchronization. The erratic behavior observed with 256 threads is explained by AMD EPYC 7763 CPUs, which have 64 cores, 128 threads. The increase in runtime across the range of particles is likely due to inefficiencies introduced by frequent context switching.





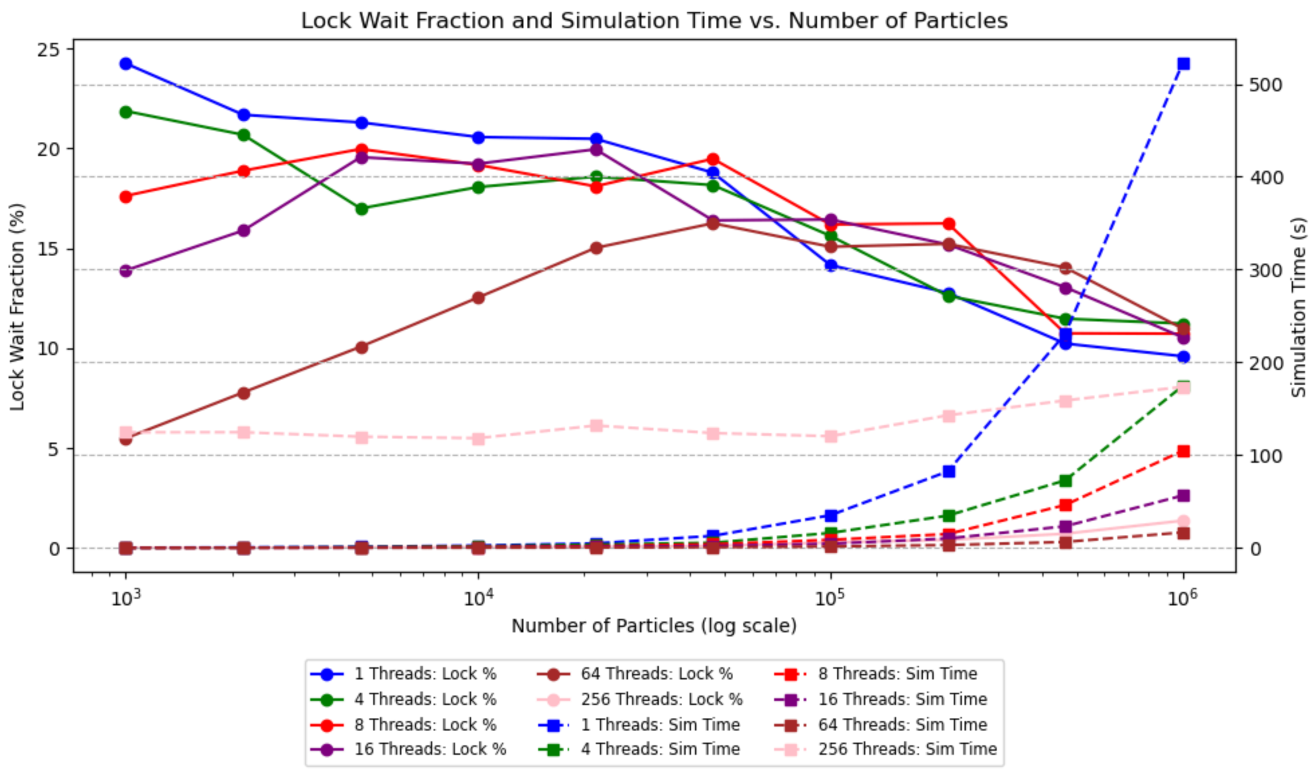
The above two plots show the strong and weak scaling behaviors of our parallel implementation. To benchmark strong scaling, we hold the number of particles constant and plot the speedup as we increase the number of threads. On a log-log scale, we would ideally like to see the curves match a line with a slope of -1. Taking a look at the strong scaling plot, we see that the trend in the midrange of thread counts (4-32 threads) approaches the slope of the ideal reference as the number of particles increases. Any difference between the empirical slopes and the ideal slope can be attributed to thread management overhead, synchronization time, and the latency of regions in the code that are not parallelized, such as init_simulation and the swap operation in simulate_one_step. In the 1000-particle line, we once again see that the benefits of multithreading are limited for smaller problem sizes. We also observe that there is significant deviation from the ideal slope in the end ranges of thread counts (namely 1-2 and 64-128 threads). When the thread count is small, the speedup from parallelizing computations is too small to outweigh the added overhead of managing threads. On the other hand, when the thread count is large, the overhead of managing these threads and time spent waiting for locks outweighs the speedup from multithreading.

To benchmark weak scaling, we increase processor count proportionally with the problem size. On a log-log scale, the ideal reference is a line with slope 0. In the weak scaling plot, we see further evidence of the trends discussed above. The ideal trend is followed in the midrange of thread counts, while there is significant deviation from the ideal reference at the end ranges. Again, the same factors explain both observations.

Regarding potential improvements to the parallel speedup, we may be able to further minimize synchronization time by designing a more fine-grained locking mechanism or iterating over bins in a more clever order to minimize contention. Additionally, we did not explore reorganizing the data to optimize cache accesses and avoid false sharing in our implementation, so some room for improvement can be found along that vector. However, we believe implementing these extensions would introduce significant complexity for an insignificant benefit in the tested problem sizes.

Computation versus Synchronization Time:

In the OpenMP-parallelized version of our simulation, the total runtime consists of two primary components: computation and synchronization. Computation time covers force calculations, force application, and particle movements, and it ideally scales inversely with the number of threads (T/p) as the workload is evenly distributed. However, synchronization time, which includes overhead from coordinating work and managing locks for bin updates, does not scale as efficiently. This overhead, particularly during particle reassignment and lock acquisition, increases with the number of threads due to contention for shared data structures. Data collected across thread counts of 1, 4, 8, 16, 64, and 256 for particle counts ranging from 1,000 to 1,000,000 suggest that while computation benefits significantly from parallelism, synchronization overhead prevents an ideal (T/p) scaling at higher thread counts. Interestingly, the data reveals that there is an optimal balance between problem size and thread count, where increasing threads beyond a certain point leads to diminishing returns due to lock contention and management overhead. This trade-off, evident in the graph, highlights the challenge of balancing workload distribution against the synchronization costs inherent in managing shared resources.


Lock Wait Fraction and Simulation Time vs. Number of Particles
Lock Wait Fraction (%)
Simulation Time (s)
Number of Particles (log scale)
10^3
10^4
10^5
10^6
1 Threads: Lock %
4 Threads: Lock %
8 Threads: Lock %
16 Threads: Lock %
64 Threads: Lock %
256 Threads: Lock %
1 Threads: Sim Time
4 Threads: Sim Time
8 Threads: Sim Time
16 Threads: Sim Time
64 Threads: Sim Time
256 Threads: Sim Time